Appendix to
Assessing the Value of Branches with What-if Analysis

March 16, 2011
Technical Report
MSR-TR-2012-33

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Appendix to
Assessing the Value of Branches with What-if Analysis

Christian Bird
Microsoft Research
Redmond, WA, USA
cbird@microsoft.com

Thomas Zimmermann
Microsoft Research
Redmond, WA, USA
tzimmer@microsoft.com

ABSTRACT
Our research paper, "Assessing the Value of Branches with What-If Analysis" contained an intuitive description of our method of creating alternative histories with branches removed and using these alternative histories along with the original history to calculate delay and provided isolation. This appendix contains a more precise formal description of our technique and metrics for the interested reader.

APPENDIX: FORMAL DEFINITIONS

A.1 Terminology
We first present definitions of the terms and ideas needed to describe branches within an SCM. We use definitions that have been accepted by the software engineering community.

File – The basic entity within an SCM is a file. We denote the set of all files tracked within the SCM as F.

Checkin – A checkin is a modification to a file via either an edit, an integration, or an anchor (all defined below). The SCM stores the contents of a file that exist after each checkin. A checkin is made on a branch, defined below. Our diagrams in this paper use a circle of some type on a horizontal line to denote a checkin on a branch. We denote the set of all checkins in an SCM as C. We also define two relations; $\boldsymbol{file}: \boldsymbol{C} \rightarrow \boldsymbol{F}$ maps a checkin to the file that the checkin affects and $\boldsymbol{time}: \boldsymbol{C} \rightarrow \mathbb{N}$ maps a checkin to the time that it was created.

Commit – A commit comprises a set of checkins to different files that acts as an atomic operation. Developers typically make changes to multiple files in the system to resolve an issue or implement a feature. They then commit all of the checkins to the SCM in batch.

Branch – A branch is a chronological ordering of subsequent checkins to a file. Checkins to the same file on different branches can be made in parallel and do not interfere with each other. Formally, a branch is an ordered sequence of distinct checkins: $\langle c_i, c_{i+1}, c_{i+2}, \cdots \rangle$. We denote the set of all branches in an SCM as B. We also define a predecessor relation $\boldsymbol{pred}: \boldsymbol{C} \rightarrow \boldsymbol{C}$ that maps a checkin to its immediately preceding checkin on the branch and $\boldsymbol{branch}: \boldsymbol{C} \rightarrow \boldsymbol{B}$ that maps a checkin to the branch that it occurs on. Despite the fact that the checkins in a branch are ordered, we use set notation for relationships between checkins and branches to avoid unnecessary complexity in notation. In our notation, the following statement holds:

$$\forall c \in C, b \in B : (branch(c) = b) \Rightarrow c \in b.$$

Root – The main branch from which all other branches are directly or indirectly created is the root branch. The root represents a special branch; we denote it formally as $r \in B$.

Parent – When one branch is created from another, the original branch is the parent of the created branch. We define a relation $\boldsymbol{parent}: \boldsymbol{B} \rightarrow \boldsymbol{B}$ that maps a branch to its parent and note that $parent(r)$ is undefined.

Child – Likewise, branches which are created from an original branch are called child branches. Checkins to files will typically be integrated from child branches to parent branches until they reach the root. We define a relation $\boldsymbol{children}: \boldsymbol{B} \rightarrow \boldsymbol{2^B}$ that maps a branch to its children (a subset of B) and a relation $\boldsymbol{desc}: \boldsymbol{B} \rightarrow \boldsymbol{2^B}$ that maps a branch to all of its descendants.

Edit – Adding a file to the SCM, modifying the contents of a file, or removing a file from SCM is an edit of that file. An edit is a type of checkin and is denoted with a solid circle. We define a predicate $\boldsymbol{edit}: \boldsymbol{C} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $edit(c)$ is true if $c \in C$ is an edit checkin.

Integration – An integration merges or copies the contents of a single file at a specific point in time on one branch into another branch. Most of the time, integrations will occur between parent and child branches, as shown in the diagram below. An integration is a type of checkin and is denoted with a large hollow circle. We define a predicate $\boldsymbol{integration}: \boldsymbol{C} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $integration(c)$ is true if $c \in C$ is an integration checkin. In addition we define the relation $\boldsymbol{source}: \boldsymbol{C} \rightarrow \boldsymbol{C}$ such that $source(c_i) = c_j$ iff $integration(c_i)$ and c_i is an integration from c_j.

Anchor – An anchor is a "placeholder" on a branch that contains no actual change to the file, but represents the state of the file at a specific point in time. When an integration occurs in an SCM, an anchor is created (see diagram below) and is the source of the integration. Anchors are denoted with a small hollow circle. We define a predicate $\boldsymbol{anchor}: \boldsymbol{C} \rightarrow \{\boldsymbol{T}, \boldsymbol{F}\}$ such that $anchor(c)$ is true if $c \in C$ is an anchor checkin. We also define a relation $\boldsymbol{target}: \boldsymbol{C} \rightarrow \boldsymbol{C}$ that maps an anchor checkin to its matching integration checkin. Note that the anchor and matching integration have the same time:

$$\Big(\text{integration}(c_i) \wedge \big(\text{source}(c_i) = c_j\big)\Big) \Rightarrow \Big(\text{anchor}(c_j) \wedge \big(\text{time}(c_i) = \text{time}(c_j)\big)\Big)$$

File History – A file history for some file, $f \in F$, is the set of all branches that contain checkins for f.

Path – A path between two checkins is a sequence of checkins that are related via $pred$ (subsequent checkins on the same branch) or $source$ (integration checkins between branches) relations. Intuitively, one can think of a path as the flow of file modifications along and between branches. Formally, $\boldsymbol{path}: \boldsymbol{C} \times \boldsymbol{C} \rightarrow \boldsymbol{2^C}$.

$$\text{path}(c_x, c_y) = \langle c_1, c_2, \cdots, c_n \rangle:$$

$$(c_1 = c_x) \wedge (c_n = c_y) \wedge ((\text{pred}(c_i) = c_{i-1}) \vee (\text{source}(c_i) = c_{i-1}))$$

A.2 Simulated Removal of a Single Branch

We define the set of checkins in the original history as C_o and the set of checkins in the alternative history as C_s. Further, we define a relation $\boldsymbol{orig}: \boldsymbol{C_S} \rightarrow \boldsymbol{C_o}$ that maps checkins in the alternative history to corresponding checkins in the original history.

First we identify all checkins that occur on the victim and move these checkins to the parent branch, while keeping chronological order. Formally, let b_{vo} be a victim branch in the original history, h_o that we are simulating the removal of, and let $b_{po} \in h_o$ be the parent in the original history. We start with an alternative history h_s that is a precise copy of h_o. That is, each branch $b_s \in h_s$ has a corresponding branch $b_o \in h_0$ and each commit $c_s \in C_s$ has a corresponding commit $c_o \in C_o$. Let b_{vs} be the victim branch that we are simulating the removal of and b_{ps} be the parent branch in the simulated history. For each edit checkin on the victim branch $c_v \in b_{vs}$, we remove c_v from b_{vs} and add it to b_{ps} while maintaining chronological predecessor relationships such that

$$((pred(c_i) = c_j) \Rightarrow (time(c_i) > time(c_j)).$$

Second, we remove all integrations between the parent and victim since they would no longer be needed. We remove all checkins $c_i \in b_{ps}$ such that

$$\begin{aligned}&\left(\text{integration}(c_i) \wedge \left(\text{branch}\left(\text{source}(c_i)\right) = b_{vs}\right)\right) \vee \\ &\left(\text{anchor}(c_i) \wedge \left(\text{branch}\left(\text{target}(c_i)\right) = b_{vs}\right)\right)\end{aligned}$$

Third, all integrations from and to the victim are redirected from the victim to the parent branch. That is, all checkins $c_i \in b_{vs}$ such that $integrate(c_i) \vee anchor(c_i)$ are changed to $branch(c_i) = b_{ps}$.

A.2 Measuring Delay & Isolation

We now present the formal definitions of delay and provided isolation that we use to measure branches.

Delay. We have already presented transit time, the time that it takes for a file change to reach the root. Since the branches within the SCM form a tree, the only way for a file modification in a checkin made on a branch to reach the rest of the system is for the modification to reach r via a series of edit, integration, and anchor checkins. Formally, the transit time for a checkin c_i is defined as:

$$\begin{aligned}&\text{transit}(c_i) = \text{minimum } t: \left(t = \text{time}(c_j) - \text{time}(c_i)\right) \wedge \\ &\left(\exists c_j \in C : \text{path}(c_i, c_j)\right) \wedge \left(\text{branch}(c_j) = r\right).\end{aligned}$$

The delay for some branch b is the average difference in transit time between the original history and the simulated history with b removed for edits that are affected by b. An edit is affected by b if the edit occurs on b or any child branches (recursively) of b. These comprise the edits whose transit time could be modified by the removal of b. Our formal definition of a checkin being affected by a branch is.

$$\text{affected(b)} = \{c \in b \cup \text{desc(b)}:\text{edit(c)}\}$$

To calculate the delay of b, we compute the transit time for each of these checkins in the original history and in the simulated history with b removed. To compute delay, we calculate the decrease in transit time for each of these checkins in the simulated history.

$$\text{delay(b)} = \sum_{c \in \text{affected(b)}} \text{transit(orig(c))} - \text{transit(c)}$$

Isolation. One value of a branch b is the isolation that it provides, which we quantify by determining the number of conflicts avoided. If there was concurrent activity on the same file in a branch and its parent or in a branch and its children, then the branch provided a level of development isolation and was useful. However, if development in a file on a branch had no potential conflicting changes in its children or parent, then this isolation was likely not needed. We calculate this by examining the order of the checkins on the parent in the simulated history and counting the number of conflicts. A conflict is a pair of subsequent edit checkins on a branch in a simulated file history that occurred on different branches in the original file history. These are be indicative of checkins that may be syntactically or semantically incompatible; even if the algorithm used by the SCM to merge textual changes runs without error, a developer must still validate that the resulting file contents does not contain problems. In our survey of Windows developers, they indicated that this often included performing a build of the system followed by running test suites of varying levels. Thus, each conflict introduced by the removal of a branch represents a non-trivial amount of additional work for a developer. We define a relation $\boldsymbol{prededit}: \boldsymbol{C} \rightarrow \boldsymbol{C}$ that maps a commit $c \in C$ to the closest preceding edit checkin. Formally we define conflict as a predicate:

$$\begin{aligned}&\text{conflict}(c_i, c_j) = \\ &\left(\text{prededit}(c_j) = c_i\right) \wedge \text{edit}(c_j) \wedge \left(\text{branch}(c_i) = \text{branch}(c_j)\right) \wedge \\ &\left(\text{branch}\left(\text{orig}(c_j)\right) \neq \text{branch}\left(\text{orig}(c_i)\right)\right)\end{aligned}$$

We therefore operationalize isolation by counting the number of conflicts that occur as a result of removing a branch. Note that conflicts are computed on a per file basis. That is, when removing a branch, we simulate the file history and count conflicts separately for each file that occurs in the branch; a checkin for `foo.c` cannot be in conflict with a checkin for `bar.h`. For space, we provide only our formal definition for isolation in the single branch removal case, the definition for other simulation types differ only in the set of branches from which the conflict checkins are drawn ($desc(b)$ in addition to b):

$$\text{isolation(b)} = \left|\{c_i, c_j \in b \cup \text{parent(b)}:\text{conflict}(c_i, c_j)\}\right|$$

While we cannot know what exactly would have actually happened had a branch not existed, our simulation effectively quantifies the isolation provided and delay introduced by a given branch. Even if developers would coordinate their changes to avoid conflicts, this *is* additional effort.

A.3 Normalizing

Some branches have an order of magnitude more changes than others. Thus total delay and total isolation may be misleading, especially when comparing different branches. As an example, branches with many edits will have more influence on total delay just because of the high number of edits. Therefore, depending on the question that we are interested in answering, delay and isolation may need to be normalized: For scenarios related to comparisons and decisions on individual branches (or subtrees), we nor-

malize the delay and isolation measures. For scenarios related branch structure as a whole, we do not normalize. More specifically, for the scenarios presented in this paper, we normalize in the following ways.

Normalized delay. The removal of branches can only affect the transit time of edits on the victim branches and on their children (recursively). We call the edits on these branches the affected edits (regardless of if their transit time is changed). Therefore, when normalizing delay, we divide the sum of the differences in transit time by the number of affected edits:

$$\text{normdelay}(b) = \frac{\sum_{c \in \text{affected}(b)} \text{transit}(\text{orig}(c)) - \text{transit}(c)}{|\text{affected}(b)|}$$

Normalized isolation. Here we normalize by the maximum number of possible conflicts that can be introduced. All edit checkins on the removed branches end up in the victims' parent branches (there may be multiple victims if multiple branch removal steps are taken from H0 to create H1). Thus, the maximum number of conflicts occurs when there is perfect interleaving of edits that were created on different branches in H0, which is the number of edits on the parent plus the number of edits on the victim minus 1: We normalize isolation by dividing the number of conflicts that the branch avoids by the maximum number of possible conflicts.

$$\text{normisolation}(b) = \frac{\left|\{c_i, c_j \in b \cup \text{parent}(b) : \text{conflict}(c_i, c_j)\}\right|}{|\{c \in b \cup \text{parent}(b) : \text{edit}(c)\}| - 1}$$

Intuitively, the normalized isolation indicates how many conflicts per edit checkin a branch prevents.